Enhancement Request for ADA

John-Eric Söderman

Summary

1. A more strict specification of &-operator
2. Wide_Wide_String literal (page 6)
3. Wide_Wide_Character literal (page 6)

Examples are based on the work with the GNAT compiler. The request is directed to the standard, because the change of the compiler should be the result of changes in the standard.

In this request the Y2018, with it's sub package "Y2018.Text" should not be considered as a part of the request for change.

Problem (1)

Facts and the standard

The "&" operator should result in an array type. Here are some not so clear questions

1. if left argument is a non-array type and right argument is an array type or a container type then the result should be an array type of the left argument (and not a container type)
2. if left argument is an array type and right argument is of the a array type or a container type then the result should be of the same array type as the left argument
3. if left argument is a container type and right argument is of the a array type or a container type then the result should be of the same container type as the left argument
4. the container types are not array types. Because you cannot specify a simple array indexing to a container type (ct.Element(I) is not the same as ct(I))

There seems to be some confusion about this in the ADA Standard package and in the GNAT library.
When dealing with "text" this confusion results in compiler and run-time errors. It is probably an error to specify in the standard packages operators, a better solution is to specify operator-only sub-packages. The programmer can choose will he or she use this operator or not, as has been done in this Enhancement Request for ADA.

GNAT implementation

Some “&”-functions violates the standard. A search of the GNAT standard library resulted in this list.

```
**** /usr/lib/gcc/x86_64-linux-gnu/4.9/rts-native/adainclude/g-spitbo.ads
0203|   function "&"(Num : Integer; Str : String)  return String;
0204|   function "&"(Str : String;  Num : Integer) return String;
0205|   function "&"(Num : Integer; Str : VString) return VString;
0206|   function "&"(Str : VString; Num : Integer) return VString;
**** /usr/lib/gcc/x86_64-linux-gnu/4.9/rts-native/adainclude/a-strunb.adb
0063|   function "&"(Left : Unbounded_String;Right : Unbounded_String) return
Unbounded_String
0103|   function "&"(Left : Unbounded_String;Right : String) return
Unbounded_String
0136|   function "&"(Left : String;Right : Unbounded_String) return
Unbounded_String [*]
0169|   function "&"(Left : Unbounded_String;Right : Character) return
Unbounded_String
0186|   function "&"(Left : Character;Right : Unbounded_String) return
Unbounded_String [*]
**** /usr/lib/gcc/x86_64-linux-gnu/4.9/rts-native/adainclude/a-stzbou.ads
0116|   function "&"(Left : Bounded_Wide_Wide_String;Right :
Bounded_Wide_Wide_String) return Bounded_Wide_Wide_String;
0120|   function "&"(Left : Bounded_Wide_Wide_String;Right : Wide_Wide_String)
return Bounded_Wide_Wide_String;
0124|   function "&"(Left : Wide_Wide_String;Right : Bounded_Wide_Wide_String)
return Bounded_Wide_Wide_String;
0128|   function "&"(Left : Bounded_Wide_Wide_String;Right :
Wide_Wide_Character) return Bounded_Wide_Wide_String;
0132|   function "&"(Left : Wide_Wide_Character;Right :
Bounded_Wide_Wide_String) return Bounded_Wide_Wide_String;
0566|   function "&"(Left : Bounded_Wide_Wide_String;Right :
Bounded_Wide_Wide_String) return Bounded_Wide_Wide_String
0571|   function "&"(Left : Bounded_Wide_Wide_String;Right : Wide_Wide_String)
return Bounded_Wide_Wide_String
0576|   function "&"(Left : Wide_Wide_String;Right : Bounded_Wide_Wide_String)
return Bounded_Wide_Wide_String
0581|   function "&"(Left : Bounded_Wide_Wide_String;Right :
Wide_Wide_Character) return Bounded_Wide_Wide_String
0586|   function "&"(Left : Wide_Wide_Character;Right :
Bounded_Wide_Wide_String) return Bounded_Wide_Wide_String
**** /usr/lib/gcc/x86_64-linux-gnu/4.9/rts-native/adainclude/a-stzunb.ads
0092|   function "&"(Left : Unbounded_Wide_Wide_String;Right :
Unbounded_Wide_Wide_String) return Unbounded_Wide_Wide_String;
0096|   function "&"(Left : Unbounded_Wide_Wide_String;Right : Wide_Wide_String)
return Unbounded_Wide_Wide_String;
0100|   function "&"(Left : Wide_Wide_String;Right : Unbounded_Wide_Wide_String)
return Unbounded_Wide_Wide_String; [*]
0104|   function "&"(Left : Unbounded_Wide_Wide_String;Right :
Wide_Wide_Character) return Unbounded_Wide_Wide_String;
0108|   function "&"(Left : Wide_Wide_Character;Right :
Unbounded_Wide_Wide_String) return Unbounded_Wide_Wide_String; [*]
**** /usr/lib/gcc/x86_64-linux-gnu/4.9/rts-native/adainclude/a-stwiun.ads
0092|   function "&"(Left : Unbounded_Wide_String;Right : Unbounded_Wide_String)
return Unbounded_Wide_String;
```

```
0096|   function "&"(Left : Unbounded_Wide_String;Right : Wide_String) return
Unbounded_Wide_String;
0100|   function "&"(Left : Wide_String;Right : Unbounded_Wide_String) return
Unbounded_Wide_String;
0104|   function "&"(Left : Unbounded_Wide_String;Right : Wide_Character) return
Unbounded_Wide_String;
0108|   function "&"(Left : Wide_Character;Right : Unbounded_Wide_String) return
Unbounded_Wide_String; [*]
**** /usr/lib/gcc/x86_64-linux-gnu/4.9/rts-native/adainclude/a-stwibo.ads
0116|   function "&"(Left : Bounded_Wide_String;Right : Bounded_Wide_String)
return Bounded_Wide_String;
0120|   function "&"(Left : Bounded_Wide_String;Right : Wide_String) return
Bounded_Wide_String;
0124|   function "&"(Left : Wide_String;Right : Bounded_Wide_String) return
Bounded_Wide_String;
0128|   function "&"(Left : Bounded_Wide_String;Right : Wide_Character) return
Bounded_Wide_String;
0132|   function "&"(Left : Wide_Character;Right : Bounded_Wide_String) return
Bounded_Wide_String;
0557|   function "&"(Left : Bounded_Wide_String;Right : Bounded_Wide_String)
return Bounded_Wide_String
0562|   function "&"(Left : Bounded_Wide_String;Right : Wide_String) return
Bounded_Wide_String
0567|   function "&"(Left : Wide_String;Right : Bounded_Wide_String) return
Bounded_Wide_String
0572|   function "&"(Left : Bounded_Wide_String;Right : Wide_Character) return
Bounded_Wide_String
0577|   function "&"(Left : Wide_Character;Right : Bounded_Wide_String) return
Bounded_Wide_String
**** /usr/lib/gcc/x86_64-linux-gnu/4.9/rts-native/adainclude/a-strbou.ads
0115|   function "&"(Left : Bounded_String;Right : Bounded_String) return
Bounded_String;
0119|   function "&"(Left : Bounded_String;Right : String) return
Bounded_String;
0123|   function "&"(Left : String;Right : Bounded_String) return
Bounded_String;
0127|   function "&"(Left : Bounded_String;Right : Character) return
Bounded_String;
0131|   function "&"(Left : Character;Right : Bounded_String) return
Bounded_String;
0555|   function "&"(Left : Bounded_String;Right : Bounded_String) return
Bounded_String
0560|   function "&"(Left : Bounded_String;Right : String) return Bounded_String
0565|   function "&"(Left : String;Right : Bounded_String) return Bounded_String
0570|   function "&"(Left : Bounded_String;Right : Character) return
Bounded_String
0575|   function "&"(Left : Character;Right : Bounded_String) return
Bounded_String
```

Because theese functions are in GNAT rts-native directory there are no way to nicely correct the problem. The fact seems to be if you use “&” operator you have to guess what will be the result.

Do’s and Do not’s

To correct the problem Y2018.Text.STR can be used in a Use-clause (“use Y2018.Text.STR;”). But do not specify **Unbounded_String** or **Unbounded_Wide_Wide_String** as a ***Use-clause***. It is OK to specify both in the With-clause.

```
with Ada.Text_IO; use  Ada.Text_IO;
with Y2018.Text; use Y2018.Text;
with Y2018.Text.STR; use Y2018.Text.STR;
with Y2018.Text.UTF; use Y2018.Text.UTF;
with Ada.Strings.Unbounded;
-- use Ada.Strings.Unbounded; << NOT THIS

procedure Fuzzy is
us:Ada.Strings.Unbounded.Unbounded_String:=Ada.Strings.Unbounded.To_Unbounded_St
ring("Urk");
begin
      Ada.Text_IO.Put_Line (UTF.To8(">" & us & "<"));
      Ada.Text_IO.Put_Line ("*** End of Fuzzy ***");
end Fuzzy;
```

This will work but is complicated. The second problem is that the GNAT compiler cannot difference between ">" as a String literal and ">" as Wide_Wide_String literal and to correct this UTF.To8 is used. In Y2018.Text package is only Wide_Wide_String used never String or Wide_String.
But by using renaming the code can be easier to code also with this use-restriction.

```
with Ada.Text_IO; use  Ada.Text_IO;
with Y2018.Text; use Y2018.Text;
with Y2018.Text.STR; use Y2018.Text.STR;
with Y2018.Text.UTF; use Y2018.Text.UTF;
with Ada.Strings.Unbounded;
-- use Ada.Strings.Unbounded; << NOT THIS

procedure Fuzzy is
package AUX renames Ada.Strings.Unbounded;
package AUW renames Ada.Strings.Wide_Wide_Unbounded;
us:AUX.Unbounded_String:=AUX.To_Unbounded_String("Urk");
begin
      Ada.Text_IO.Put_Line (UTF.To8(">" & us & "<"));
      Ada.Text_IO.Put_Line ("*** End of Fuzzy ***");
end Fuzzy;
```

We still have here the problem with the ADA-compiler. In this case it seems that the compiler chooses to use Wide_Wide_String for ">" and "<", maybe that the compiler tries to honor the UTF.To8 choice and then the overloading of the ampersand is successful in this case.

The STR style

In using Y2018.Text package you have a choice

1. Use Y2018.Text.STR
2. Do not use the "Y2018.Text.STR" but use Ada.Strings.Unbounded and Ada.Wide_Wide_Strings.Unbounded

If you decide to use the STR then you can write {something} & {something} & {something} & … and be sure that the result of the whole expression is of the same type as the first "something" except if the first "something" is a Wide_Wide_Character then the result is a Wide_Wide_String. This feature is not possible in the second case. Inside the STR overloading package Bounded_String and Bounded_Wide_Wide_String is not supported. In STR and other Y2018.Text sub-packages only Wide_Wide_String is defined which makes it clear for the ADA compiler that the arguments cannot be String or Wide_String and the get rid of the multiple choice errors for strings.

The STR style comes with a cost, you have to spell out Ada.Strings.Unbounded and Ada.Wide_Wide_Strings.Unbounded.
Inside the Y2018.Text.STR package following specifications are defined:

```
1. function "&"( left:Wide_Wide_String; right:Wide_Wide_Character) return
   Wide_Wide_String;
2. function "&"( left:Wide_Wide_String; right:Unbounded_Wide_Wide_String)
   return Wide_Wide_String;
3. function "&"( left:Wide_Wide_String; right:Unbounded_String) return
   Wide_Wide_String;
4. function "&"( left:Unbounded_Wide_Wide_String; right:Wide_Wide_String)
   return Unbounded_Wide_Wide_String;
5. function "&"( left:Unbounded_Wide_Wide_String;
   right:Unbounded_Wide_Wide_String) return Unbounded_Wide_Wide_String;
6. function "&"( left:Unbounded_Wide_Wide_String; right:Unbounded_String)
   return Unbounded_Wide_Wide_String;
7. function "&"( left:Unbounded_String; right:Wide_Wide_String) return
   Unbounded_String;
8. function "&"( left:Unbounded_String; right:Unbounded_Wide_Wide_String)
   return Unbounded_String;
9. function "&"( left:Unbounded_String; right:Unbounded_String) return
   Unbounded_String;
10.      function "&"( left:Unbounded_Wide_Wide_String;
   right:Wide_Wide_Character) return Unbounded_Wide_Wide_String;
11.      function "&"( left:Unbounded_String; right:Wide_Wide_Character)
   return Unbounded_String;
12.      function "&"( left:Wide_Wide_Character; right:Wide_Wide_String)
   return Wide_Wide_String;
13.      function "&"( left:Wide_Wide_Character; right:Wide_Wide_Character)
   return Wide_Wide_String;
14.      function
   "&"(left:Wide_Wide_Character;right:Unbounded_Wide_Wide_String) return
   Wide_Wide_String;
15.      function "&"(left:Wide_Wide_Character;right:Unbounded_String) return
   Wide_Wide_String;
1.
```

Note the omission of function "&"(left:Wide_Wide_String;right:Wide_Wide_String) return Wide_Wide_String; because this is a part of basic ADA.
Following table maybe helpful:

left\right argument	**Wide_Wide_String**	**Unbounded_Wide_ Wide_String**	**Unbounded_String**	**Wide_Wide_Charact er**
Wide_Wide_String	(0) Wide_Wide_String	2 Wide_Wide_String	3 **[UTF-32]** Wide_Wide_String	1 Wide_Wide_String
Unbounded_Wide_ Wide_String	4 Unbounded_Wide_Wi de_String	5 Unbounded_Wide_Wi de_String	6 **[UTF-32]** Unbounded_Wide_Wi de_String	10 Unbounded_Wide_Wi de_String
Unbounded_String	7 **[UTF-8]** Unbounded_String	8 **[UTF-8]** Unbounded_String	9 Unbounded_String	11 **[UTF-8]** Unbounded_String
Wide_Wide_Charact er	12 Wide_Wide_String	14 Wide_Wide_String	15 **[UTF-32]** Wide_Wide_String	13 Wide_Wide_String

In every table element is the result type of the expression. Some of functions contains UTF-32 or UTF-8 conversions.

With Wide_Wide_String as a different type than String

Same table but extended with rows and columns for String and Character:

left\right argument	**String**	**Wide_Wide_String**	**Unbounded_Wide_Wide_String**	**Unbounded_String**	**Character**	**Wide_Wide_Character**
String	String	**[UTF-8]** String	**[UTF-8]** String	String	String	**[UTF-8]** String
Wide_Wide_String	**[UTF-32]** Wide_Wide_String	(0) Wide_Wide_String	2 Wide_Wide_String	3 **[UTF-32]** Wide_Wide_String	**[UTF-32]** Wide_Wide_String	1 Wide_Wide_String
Unbounded_Wide_Wide_String	**[UTF-32]** Unbounded_Wide_Wide_String	4 Unbounded_Wide_Wide_String	5 Unbounded_Wide_Wide_String	6 **[UTF-32]** Unbounded_Wide_Wide_String	**[UTF-32]** Unbounded_Wide_Wide_String	10 Unbounded_Wide_Wide_String
Unbounded_String	Unbounded_String	7 **[UTF-8]** Unbounded_String	8 **[UTF-8]** Unbounded_String	9 Unbounded_String	Unbounded_String	11 **[UTF-8]** Unbounded_String
Character	String	**[UTF-8]** String	**[UTF-8]** String	String	String	**[UTF-8]** String
Wide_Wide_Character	**[UTF-32]** Wide_Wide_String	12 Wide_Wide_String	14 Wide_Wide_String	15 **[UTF-32]** Wide_Wide_String	**[UTF-32]** Wide_Wide_String	13 Wide_Wide_String

… but this cannot be specified in ADA today.

Problem (2 and 3)

GNAT ADA compiler generates String, Wide_String and Wide_Wide_String from any String literal it sees and the GNAT runtime handles String, Wide_String and Wide_Wide_String as ”the same type”. This makes writing Unicode letters a problem if the letter is not in basic ASCII.

UTF-32BE seems to be the best candidate for Unicode text and with simple mapping to Wide_Wide_String. GNAT input source code is in UTF-8 (in our case) and with the problems with String, Wide_String and Wide_Wide_String literals, literal handling is extended with a new literals, literals for Wide_Wide_String and for Wide_Wide_Character. These extensions are ”What”W as a Wide_Wide_String literal and ’W’W as a Wide_Wide_Character literal. For every ”What”W literal a Wide_Wide_String constant is generated and filled with the text as numeric code point values and for every ’W’W a Wide_Wide_Character constant is generated with a numeric code point value.

The result for writing code is that you can code ”This is true”W instead of writing (Wide_Wide_Character'Val(16#54#), Wide_Wide_Character'Val(16#68#), Wide_Wide_Character'Val(16#69#), Wide_Wide_Character'Val(16#73#), Wide_Wide_Character'Val(16#20#), Wide_Wide_Character'Val(16#69#), Wide_Wide_Character'Val(16#73#), Wide_Wide_Character'Val(16#20#), Wide_Wide_Character'Val(16#74#), Wide_Wide_Character'Val(16#72#), Wide_Wide_Character'Val(16#75#), Wide_Wide_Character'Val(16#65#)). Of course these characters are all basic ASCII but if the text contains characters outside basic ASCII then you have this complication.

The text ”Detta är en lögn”W (in Swedish, this is a lie) is

(Wide_Wide_Character'Val(16#44#), Wide_Wide_Character'Val(16#65#),
Wide_Wide_Character'Val(16#74#), Wide_Wide_Character'Val(16#74#),
Wide_Wide_Character'Val(16#61#), Wide_Wide_Character'Val(16#20#),
Wide_Wide_Character'Val(16#E4#), Wide_Wide_Character'Val(16#72#),
Wide_Wide_Character'Val(16#20#), Wide_Wide_Character'Val(16#65#),
Wide_Wide_Character'Val(16#6E#), Wide_Wide_Character'Val(16#20#),
Wide_Wide_Character'Val(16#6C#), Wide_Wide_Character'Val(16#F6#),
Wide_Wide_Character'Val(16#67#), Wide_Wide_Character'Val(16#6E#)).

To make it more complicated the text ”euro €”W is
(Wide_Wide_Character'Val(16#65#), Wide_Wide_Character'Val(16#75#),
Wide_Wide_Character'Val(16#72#), Wide_Wide_Character'Val(16#6F#),
Wide_Wide_Character'Val(16#20#), Wide_Wide_Character'Val(16#20AC#)).

W-String literal content is
`([^"\\]|\"|\'|\\|\b|\f|\n|\r|\t|\x[0-9A-Fa-f]{2}|\u[0-9A-Fa-f]{4}|\v[0-9A-Fa-f]{6}|\y[0-9A-Fa-f]{8}|"")*`
and W-Character literal content is
`[^'\\]|\"|\'|\\|\b|\f|\n|\r|\t|\x[0-9A-Fa-f]{2}|\u[0-9A-Fa-f]{4}|\v[0-9A-Fa-f]{6}|\y[0-9A-Fa-f]{8}|''`
as regular expression patterns.

- [^”\\] in String and [^’\\] in Character – any other except backslash and quotation mark or apostrophe
- \” – escaped quotation mark
- \’ – escaped apostrophe
- \b – generate BS
- \f – generate FF
- \n – generate LF
- \r – generate CR
- \t – generate HT
- \x[0-9A-Fa-f]{2} – generate Wide_Wide_Character corresponding to this hexadecimal value (two digits)
- \u[0-9A-Fa-f]{4} – generate Wide_Wide_Character corresponding to this hexadecimal value (four digits)
- \v[0-9A-Fa-f]{6} – generate Wide_Wide_Character corresponding to this hexadecimal value (six digits)
- \y[0-9A-Fa-f]{8} – generate Wide_Wide_Character corresponding to this hexadecimal value (eight digits)

Notation \x[0-9A-Fa-f]{2} we have in C and Perl. Notation \u[0-9A-Fa-f]{2} we have in Java. Notation \v[0-9A-Fa-f]{6} is new and can be used for Unicode characters which are at most 21-bits. Notation \y[0-9A-Fa-f]{8} is also new but usage is not clear (today) but consistent. Because of the Wide_Wide_Character limit, maximum value is \y7FFFFFFF in ADA, values outside the limit is considered to be an error. The hexadecimal values are, as in ADA, is in Big-Endian format. The reason to use y-character as the marker is to use something different than ”\w”, which is the meta character code for word character in a regular expression pattern.

A:Wide_Wide_String:="A good reason\nHappy New Year\nRegards and 10 € for \v010600"W;

A:Wide_Wide_String:=(Wide_Wide_Character'Val(16#41#),Wide_Wide_Character'Val(16#20#),
Wide_Wide_Character'Val(16#67#),Wide_Wide_Character'Val(16#6F#),Wide_Wide_Character'Val
(16#6F#),Wide_Wide_Character'Val(16#64#),Wide_Wide_Character'Val(16#20#),Wide_Wide_Cha

```
racter'Val(16#72#),Wide_Wide_Character'Val(16#65#),Wide_Wide_Character'Val(16#61#),Wide_
Wide_Character'Val(16#73#),Wide_Wide_Character'Val(16#6F#),Wide_Wide_Character'Val(16#6E
#),Wide_Wide_Character'Val(16#0A#),Wide_Wide_Character'Val(16#48#),Wide_Wide_Character'
Val(16#61#),Wide_Wide_Character'Val(16#70#),Wide_Wide_Character'Val(16#70#),Wide_Wide_
Character'Val(16#79#),Wide_Wide_Character'Val(16#20#),Wide_Wide_Character'Val(16#4E#),Wi
de_Wide_Character'Val(16#65#),Wide_Wide_Character'Val(16#77#),Wide_Wide_Character'Val(16
#20#),Wide_Wide_Character'Val(16#59#),Wide_Wide_Character'Val(16#65#),Wide_Wide_Charact
er'Val(16#61#),Wide_Wide_Character'Val(16#72#),Wide_Wide_Character'Val(16#0A#),Wide_Wid
e_Character'Val(16#52#),Wide_Wide_Character'Val(16#65#),Wide_Wide_Character'Val(16#67#),
Wide_Wide_Character'Val(16#61#),Wide_Wide_Character'Val(16#72#),Wide_Wide_Character'Val(
16#64#),Wide_Wide_Character'Val(16#73#),Wide_Wide_Character'Val(16#20#),Wide_Wide_Char
acter'Val(16#61#),Wide_Wide_Character'Val(16#6E#),Wide_Wide_Character'Val(16#64#),Wide_W
ide_Character'Val(16#20#),Wide_Wide_Character'Val(16#31#),Wide_Wide_Character'Val(16#30#)
,Wide_Wide_Character'Val(16#20#),Wide_Wide_Character'Val(16#20AC#),Wide_Wide_Character'
Val(16#20#),Wide_Wide_Character'Val(16#66#),Wide_Wide_Character'Val(16#6F#),Wide_Wide_
Character'Val(16#72#),Wide_Wide_Character'Val(16#20#),Wide_Wide_Character'Val(16#010600#
));
```

UTF and Character, Wide_Character, Wide_Wide_Character

Unicode Encoding Forms

Unicode uses five encoding forms

- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-32BE
- UTF-32LE

and bits needed for storing a Unicode character (code point) are 21. The result is that only UTF-32BE and UTF-32LE supports one character in one storage unit.

The UTF-8 problem

UTF-8 as the base for Unicode characters and mapping this to ADA Character results in problems with indexing characters and the length of a character.
The GNAT ADA compiler uses UTF-8 coded data as input. UTF-8 is well packed Unicode and saves storage space. But from a programming viewpoint it is hard to handle with a character length between one to four bytes (Unicode UTF-8 encoding).

The UTF-16 encoding forms have the same problem as UTF-8. In UTF-16 a character is stored in one or two 16 bits storage units, and is not as well packed as UTF-8.

Solution UTF-32BE

UTF-32BE maps Unicode code point to Wide_Wide_Character one character to one storage unit. If the platform do not support UTF-32BE but rather UTF-32LE a mapping as in ADA should be done to Big-Endian notation in the source code. UTF-32 is also used when defining code points for Unicode characters (Values in memory is stored as UTF-32LE on an Intel platform, but the value is written in ADA as UTF-32BE with hexadecimal digits).

ADA do not have a elegant literal notation for Wide_Wide_Characters and this must be created. A solution to this is a notation “<text>”W for Wide_Wide_String literals and ‘<character>’W for Wide_Wide_Characters (see page 6) (This type of solution can be found in PL/I). There are no reason to not implement the well known escape sequences from Java, C and Perl languages inside a W-string or a W-character literal.

With the Unicode code point mapped to Wide_Wide_String or what is a Wide_Wide_Character array type, we can use ADA array slicing to get parts of the array and we can also assign text to parts of the array. Then by using the Y2018.Text.STR sub-package we do have Unbounded_String, Unbounded_Wide_Wide_String and Wide_Wide_String as a well integrated unit and can mix text from these types in one expression (see page 1). In many cases no parenthesis or conversion functions is needed in the source code.

The Y2018.Text package

The Y2018.Text sub-package contains (sub-) sub-packages

- STR – contains overloading of “&” operator, nothing more or nothing less (see page 1).
- UTF – conversions from Wide_Wide_String to String (To8) and the reverse conversion (To32).

The Y2018.Text package do not contain any procedures, functions, as result do not have a body.